



Calhoun: The NPS Institutional Archive

Faculty and Researcher Publications

Faculty and Researcher Publications Collection

2008

Open-DIS: an open source implementation of the DIS Protocol for C++ and Java

McGregor, Don

<http://hdl.handle.net/10945/47799>



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

Open-DIS: An Open Source Implementation of the DIS Protocol for C++ and Java

Don McGregor

Don Brutzman

Naval Postgraduate School, MOVES Institute.

700 Dyer Road, Bldg 246, Rm 265

Monterey, CA 93943

831-656-2149

{mcgredo | brutzman }@nps.edu

John Grant

Systems Technology, Inc

jgrant@systemstech.com

Keywords:

Distributed Interactive Simulation, DIS, Open-DIS, Open Source

ABSTRACT: *The Distributed Interactive Simulation (DIS) protocol has long been used in military simulations, but no widespread open source C++ implementation has been made available to date. We have written an open source implementation in C++ and Java called Open-DIS that we believe will result in a less duplicated effort in creating DIS simulations. The Open-DIS Java and C++ implementations were created from an XML template language, and other language implementations can be created as needed. The C++ and Java implementations are capable of representing the data contained in protocol data units in DIS format, and the Java implementation can in addition represent the data in XML or Java object serialization formats. An XML schema for the XML format is also provided. The project is royalty-free, open source and has a non-viral Berkeley Software Distribution (BSD) license. We describe the approach taken in creating the implementation, the outlines of the implementation, and the community established for its maintenance and improvement.*

1. Distributed Interactive Simulation

The Distributed Interactive Simulation protocol (DIS, IEEE-1278.x) [1] is a standard for binary exchange of information in military simulations. The interoperability of the standard relies primarily on a consistent format for information on the wire and agreed-upon enumerations of constant values. Anyone can obtain the IEEE-1278.x standard and implement a compliant DIS library using whatever programming language API they choose.

The Modeling, Virtual Environments, and Simulation (MOVES) Institute at the Naval Postgraduate School (NPS) has in the past implemented portions of the DIS protocol in Java, using a variety of techniques. DIS-XML [2] used an XML schema and Sun's Java for XML Binding (JAXB) [3] to automatically generate code that could marshal and unmarshal data to XML, and added hand-written code to marshal and unmarshal from the IEEE-1278.x binary format. This worked well, and the ability

to marshal to XML format was found to be particularly useful. We have used Extensible Messaging and Presence Protocol (XMPP) chat servers and DIS Protocol Data Units (PDUs) in XML format to bridge wide area networks across firewalls [2], and have used XML as an archiving format for DIS traffic. We have made use of DIS-XML in other projects, such as AUV Workbench [4] and Xj3D [5] with good results, and have made use of the Java loading facility within Mathwork's MATLAB product to make use of DIS inside of MATLAB. The drawbacks to the DIS-XML approach were primarily on the development side and included the necessity of first writing the XML schema, and then having to write the code to marshal and unmarshal the information to IEEE-1278.x format to complete the code generated by JAXB. While JAXB is a useful tool for creating Java code from XML schemas, similar tools for C++ were found wanting, either because they required a license or because the code they generated was thought to be impenetrable by typical programmers. The API generated was also not necessarily similar to

the API of the Java code. It was thought that a similar API for both Java and C++ would shorten programmer learning curves.

A C++ implementation was considered to be very useful, so the problems encountered in generating C++ code from XML schema were disappointing. Most of the C++ DIS implementations to date have either been commercial products or internal to a specific corporate project. Both of these approaches have drawbacks. Commercial products often have good performance and support, but some DIS modeling and simulation projects are unwilling to encumber their efforts with a dependency on a commercial license, in part because of the very long software lifecycles in Department of Defense (DoD) projects. A project that lasts for twenty years may outlast some of the companies providing commercial software licenses and leave the project in limbo or at least in need of a major rewrite in order to replace the no-longer available commercial code. Internal project DIS implementations very often wind up duplicating previous work rather than using programmer time to add value and innovative features. We want those implementing DIS projects to stop working on the DIS and start working on the project.

2. Open-DIS

Open-DIS is an effort to make available to the modeling and simulation community a full implementation of the DIS protocol in both C++ and Java. It is open source and carries a Berkeley Software Distribution (BSD) license. The BSD license is notable for its non-viral nature; there is no requirement for any modifications written by users to be released, and code that uses the library does not become open source. Users can simply make use of and modify the code as they see fit, while retaining all rights to the code they write. Corrections or additions to the Open-DIS implementation are welcome, however.

Open-DIS is a cooperative effort to create a complete and correct DIS implementation. It consists of

- An abstract description of the DIS Protocol Data Units (PDUs) in XML
- C++ and Java language implementations of the PDUs
- Code to marshal and unmarshal the Open-DIS objects to various formats

- An XML schema for the DIS protocol
- A community to support and extend the implementation.

2.1 Abstract description of PDUs using XML

The PDUs of the DIS specification consist of a few dozen discrete binary data packets formatted in accordance with the DIS specification. The PDUs contain fields for information such as an entity's velocity and position. The exact location and format of the fields within the packet is defined by the DIS standard.

In C++, Java, and most other object-oriented languages, a PDU can be modeled as a class that has:

- Instance variables for each field in a PDU
- Methods that get and set each field value
- A constructor
- Methods to marshal and unmarshal data from the binary DIS data format.

If we begin with an abstract description of the fields in a DIS PDU that includes their type and position in the PDU, that description can be processed to generate whatever computer language code is desired. Multiple computer language implementations can be generated from a single abstract description. For example, if we have a description that specifies a DIS Entity State PDU has an entity location field of three double-precision floats, followed by a velocity field that has three single-precision floats, we can use this information to generate a source code implementation in either Java or C++ that includes at least the minimum attributes described above. A Java or C++ implementation of the DIS PDUs is about 30,000 lines of code, so creating a template from which we generate programming language code can save a significant amount of programming time.

We chose to use a self-defined XML dialect to create the abstract description of PDUs rather than the XML schema used in DIS-XML. The XML dialect we used is simpler than XML schema, and therefore easier to parse and process. It also contains in some respects more information about the DIS protocol than XML schema, which allows the creation of code that more closely resembled what would be created in a hand-written implementation. The protocol description language ASN.1 also could have been used, but we preferred to retain complete

control of the source code generated, and an XML dialect seemed a more direct path to that goal.

The DIS standard describes, mostly, four types of fields in PDUs:

- Primitive values: integers of various lengths, signed or unsigned, and floating point values of either single precision or double precision.
- Another, smaller subunit of the PDU that is treated as a reusable object
- An array of fixed length
- A list of variable length

Primitive values are single scalar values. The objects contained within PDUs include fields such as the velocity, which consists of three float values that are treated as a single object. They contain multiple values contained within one field name. Arrays of fixed length are used for character and data arrays. Variable length lists contain zero or more repetitions of an element; they are typically preceded (not necessarily immediately) by a field that holds a count of how many items are in the list. Variable length lists are one area where a custom XML dialect for describing DIS is superior to XML schema. XML schema does not have a way for meta-information to be easily added to the schema to describe the relationship between the count field and the variable length list field, while this operation is straightforward in a custom XML dialect. The meta-information can be used to generate better language implementation code.

The DIS specification describes the PDUs in a way that is amenable to description using an object-oriented inheritance hierarchy. Section 5.3.2 of IEEE-1278.1 describes PDUs and their grouping into “PDU families.” For example, the Entity State PDU (ESPDU) and Collision PDU are both members of the Entity Information/Interaction PDU family.

The ESPDU and Collision PDUs share several fields; this can be modeled using standard object-oriented techniques by using a common abstract superclass of PDU, an intermediate level abstract class of EntityInformationPdu that inherits from Pdu, and two concrete subclasses classes, EntityStatePdu and CollisionPdu, that inherit from EntityInformationPdu.

Figure 1 shows a simple example of the XML description file, in this case an excerpt from the PDU header that is common to all PDUs.

```
<class name="Pdu"
inheritsFrom="root" comment="The
superclass for all PDUs.">

  <attribute
name="protocolVersion"
comment="The version of the
protocol. 5=DIS-1995, 6=DIS-
1998.">
    <primitive type= "unsigned
byte" defaultValue="6"/>
  </attribute>

  <attribute name="exerciseID"
comment="Exercise ID">
    <primitive type= "unsigned
byte" defaultValue="0"/>
  </attribute>

  <attribute name="pduType"
comment="Type of pdu, unique for
each PDU class">
    <primitive type="unsigned
byte"/>
  </attribute>
...
```

Figure 1. Excerpt From PDU Description File

The XML dialect is straightforward. The comment attribute can be extracted to supply comments to the generated language source code, and the default value can be used to specify the value to which the fields should be set in the constructor. The “inheritsFrom” attribute of the class element can be used to define the class inheritance hierarchy described earlier. The abstract description of the DIS standard is written by a programmer reading the standard and writing some XML that describes each PDU.

2.2 Java and C++ Language Implementations.

The XML description of the DIS PDUs was processed by a small program to generate Java and C++ source code. Enough information is present in the XML file to create an implementation that is nearly as complete as hand-written code. The generated code does not, however, completely and correctly describe every PDU. The abstract description was unable to correctly describe about fifteen of the 65 or so

PDU. The work remaining to be done on those PDUs ranges from minor to significant. The defects in the generated code do not necessarily have to be fixed in the generator. The generated code is an artifact in its own right, and can be checked into source code control for manual modification, rather than attempting to get the code generator exactly right.

As mentioned earlier, the PDUs generated by Open-DIS are arranged in a class inheritance hierarchy that is the same for C++ and Java. A portion of the class inheritance diagram that includes the Entity Information/Interaction PDU family is shown in figure 2.

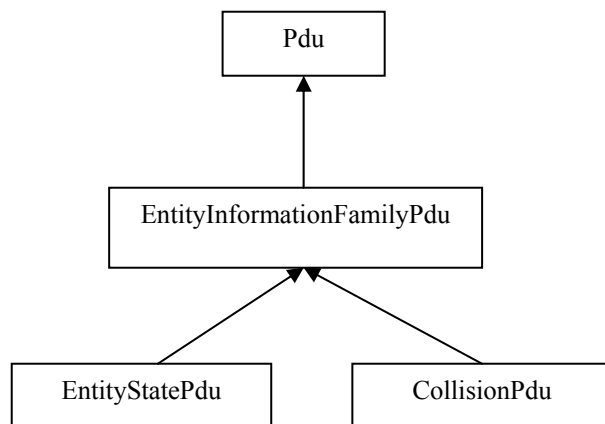


Figure 2. Partial DIS Class Inheritance Hierarchy for Entity Information/Interaction Family

The other PDU families are arranged in a similar inheritance hierarchy. All PDUs eventually inherit from the Pdu class.

A list of the implemented PDUs organized by PDU family is shown in figure 3. The PDUs that are not complete or not yet fully implemented are shown in bold.

Pdu:

EntityInformationFamilyPdu: EntityStatePdu, FastEntityStatePdu, CollisionElasticPdu, CollisionPdu, EntityStateUpdatePdu

WarfareFamilyPdu: FirePdu, DetonationPdu

LogisticsFamilyPdu: ServiceRequestPdu, ResupplyOfferPdu, ResupplyReceivedPdu, ResupplyCancelPdu, RepairCompletePdu, RepairResponsePdu

SimulationManagementFamilyPdu: StartResumePdu, StopFreezePdu, AcknowledgePdu, ActionRequestPdu, ActionResponsePdu, DataQueryPdu, SetDataPdu, DataPdu, EventReportPdu, CommentPdu, CreateEntityPdu, RemoveEntityPdu

DistributedEmissionsFamilyPdu:

ElectronicEmissionsPdu, DesignatorPdu, **UaPdu**, IffAtcNavAidsLayer1Pdu, IffAtcNavAidsLayer2Pdu, SeesPdu

RadioCommunicationsFamilyPdu:

TransmitterPdu, **SignalPdu**, ReceiverPdu, IntercomSignalPdu, IntercomControlPdu

MinefieldFamilyPdu: MinefieldStatePdu,

MinefieldQueryPdu, **MinefieldDataPdu**, MinefieldResponseNackPdu

EntityManagementFamilyPdu:

AggregateStatePdu, **IsGroupOfPdu**, **TransferControlRequestPdu**, IsPartOfPdu

SyntheticEnvironmentFamilyPdu:

EnvironmentalProcessPdu, **GriddedDataPdu**, PointObjectStatePdu, LinearObjectStatePdu, ArealObjectStatePdu

SimulationManagementWithReliabilityFamilyPdu:

StartResumeReliablePdu, StopFreezeReliablePdu, AcknowledgePduReliablePdu, ActionRequestReliablePdu, ActionResponseReliablePdu, DataQueryReliablePdu, **SetDataReliablePdu**, DataReliablePdu, **EventReportReliablePdu**, **CommentReliablePdu**, CreateEntityReliablePdu, RemoveEntityReliablePdu, RecordQueryReliablePdu, **SetRecordReliablePdu**, RecordReliablePdu

Figure 3. Open-DIS PDUs

Both the Java and C++ code are generated with class definitions, instance variables for each of the fields, methods to get and set the field values, and methods to marshal the objects to IEEE-1278.x format. The Java objects can also marshal themselves to XML or Java object serialization format. Supporting objects for fields contained within the PDUs are also generated. For example, the EntityID data structure is used throughout DIS to uniquely identify an entity in

the world. It consists of three short integer values, each one of which is a unique numeric identifier for the site, application, and entity. This triplet is defined in Open-DIS as an object in its own right, and is often used in get() and set() methods. Likewise, values such as the entity velocity can be represented by a Vector3Float object, which represents a vector with single-precision floating point fields for x, y, and z.

A Unified Modeling Language (UML) diagram for the Java PDU class is shown in figure 3.

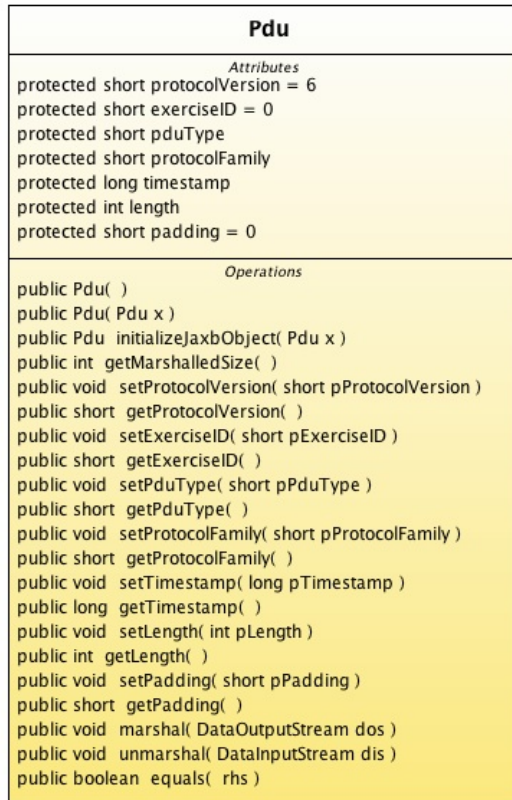


Figure 3. UML Diagram for Pdu Class

The instance variables correspond to the defined fields in the IEEE-1278.x specification for the PDU header. All PDUs inherit from this class. The marshal() and unmarshal() methods are used to convert the Java or C++ object to and from the binary format specified by the IEEE 1278.x standard.

The C++ code is very similar to the Java code. The C++ implementation includes separate header (.h) and implementation (.cpp) files, a destructor method to reclaim dynamically allocated memory, and an additional “const” get

method, which returns a non-modifiable reference to PDU fields.

A UML class diagram for the more complex entity state PDU is shown in figure 4.



Figure 4. UML diagram for Entity State PDU

The EntityStatePdu class demonstrates the use of objects to get and set fields such as the entity location. The entity location field is defined as a Vector3Double, ie a vector containing three double precision floating point values. The EntityStatePdu class also overrides some methods, such as the marshal and unmarshal methods. Internally these methods call the superclass first to marshal the fields contained in classes EntityStatePdu inherits from. The class also allows access to all the public methods of classes it inherits from, including the Pdu class discussed earlier.

Java uses garbage collection (GC) to reclaim memory used in objects that are no longer referenced, in contrast to the manual memory management typically used in C++ programs. However, GC can be problematic in real time applications because it can impose an unexpected and significant CPU load on the host that is running the program. Packets may arrive at a high rate and each of the PDUs and the objects that are contained within them, such as the entity location field described above, will eventually have to be found by the GC subsystem so their memory can be freed. In this

situation it makes sense to minimize the number of Java objects generated.

The Java PDUs contain fields that are themselves objects, such as the entity location field discussed above. While desirable from a software engineering and programmer convenience standpoint, this also increases the number of Java objects generated, which places a greater load on the Java GC subsystem when the memory must be reclaimed.

Research has shown that the vast majority of PDUs in a typical DIS exercise are Entity State PDUs. [6] If the Entity State PDUs are made more memory-efficient, most of the garbage being generated will be eliminated while the programmer will retain a convenient API for the other PDUs. To this end we have also provided a stripped-down “Fast ESPDU”, which contains only the bare minimum possible of fields represented as objects. All fields in the PDU are represented as primitive types that do not require allocation on the Java heap. In high performance implementations it may be worthwhile to use this alternative implementation.

2.3. Marshalling and Unmarshalling

The Open-DIS project also contains supporting code that allows programmers to marshal and unmarshal PDUs to several formats of their choosing, including IEEE-1278.x, XML, and Java object serialization in the case of the Java implementation. The C++ implementation marshals to only the IEEE-1278.x format.

Almost any DIS application will need to read IEEE-1278.x information from the network and convert it into an accessible format for internal application use. The Java code uses the PduFactory class to unmarshal PDUs. PduFactory implements the “Gang of Four” [7] factory object pattern. A UML diagram of PduFactory is shown in Figure 5.

PduFactory
<i>Attributes</i> private boolean useFastPdu = false
<i>Operations</i> public PduFactory() public PduFactory(boolean useFastPdu) public Pdu createPdu(byte data[0..*])

Figure 5. UML Diagram for PduFactory

The factory createPdu() method accepts an array of bytes in IEEE-1278.x format and returns a concrete PDU subclass that corresponds to that byte array. For example, a network socket may read an array of bytes from the network; we pass this to the method createPdu() in an instance of the PduFactory class and receive back an instantiated Pdu object. The Pdu instance will be a concrete subclass such as EntityStatePdu, CollisionPdu, or FirePdu. The exact class returned can be determined at runtime in Java by the instanceof operator, or by calling the getPduType() method to return the appropriate enumerated value in either Java or C++. The process of converting the byte array to a Pdu object instance is hidden entirely inside the createPdu() method. To create Entity State PDUs that create minimal amounts of garbage, use the alternate PduFactory constructor with the Boolean argument. This instantiates a factory that returns FastEntityStatePdu objects rather than EntityStatePdu objects, which will generate less garbage for realtime applications.

The C++ code uses a similar factory pattern to create new PDU objects from byte arrays.

The Open-DIS Java implementation is capable of marshalling PDU objects to IEEE-1278.x format, XML, or Java object serialization format. Figure 6 illustrates code that marshals Java language PDUs to IEEE-1278.x format.

```
EntityStatePdu espdu =
    new EntityStatePdu();
ByteArrayOutputStream baos =
    new ByteArrayOutputStream();
DataOutputStream dos =
    new DataOutputStream(baos);
espdu.marshal(dos);
byte[] ieeeData =
    baos.toByteArray();
```

Figure 6. Marshalling Java PDUs to IEEE-1278.x Format

A typical use for this code would be to create a new PDU, set field values such as location, orientation and velocity, and then place it into IEEE-1278.x format and send it to other hosts over the network.

Each PDU object and all the objects contained within them also implement the Java Serializable

interface, which allows them to be marshaled to the standard Java `ObjectStream` class. An example of this is shown in figure 7.

```
EntityStatePdu espdu =  
    new EntityStatePdu();  
FileOutputStream fos =  
    new FileOutputStream("pdus");  
fos.writeObject(espdu);
```

Figure 7. Marshalling Open-DIS PDUs to Java Object Serialization Format

This marshals the information contained in the entity state Pdu to an alternative binary format supported by Sun Microsystems. The ability to send PDU objects in the Java object stream format can be useful when communicating with other Java programs or Java-specific tools.

The ability to marshal Open-DIS objects to XML was more difficult to implement. There is no agreed-upon standard for an XML representation of DIS. Creating an XML schema is in itself a difficult and error-prone process, and one that is also a profitable area for SISO standardization efforts. Instead of writing a schema from scratch or reusing the partial schema we developed in DIS-XML, we used the JAXB 2.x release, which is capable of processing JavaBeans-compliant code to generate an XML schema that matches the code.

JavaBeans-compliant code conforms to a few simple coding standards, such as having methods that follow the idiom `getXXX()` and `setXXX()` for every field named XXX. Since the Open-DIS code was generated with the JavaBeans standard in mind, it was a simple matter to run the JAXB tool called “schemagen” on the existing source code to generate an XML schema that exactly matched the Open-DIS Java implementation. The schema is in itself a significant product for the effort, since it provides a nearly complete XML schema for DIS.

Once the schema is in hand we have many options. Keeping in mind that our ultimate objective was to be able to marshal to and from XML without having to spend weeks writing the code, we noted that JAXB generates code that does precisely this, given a schema. Somewhat counter-intuitively, we used the schema we generated from the original Open-DIS code to generate a parallel JAXB implementation of Open-DIS in a separate Java package. The Java

code generated by JAXB has the same class names, fields, and the same names for accessor methods, but the JAXB-generated classes in this alternate package can marshal themselves to and from XML; this capability was provided by JAXB.

At this point we have two classes for each PDU: one in the Open-DIS java package that can marshal itself to IEEE-1278.x format, and one in the JAXB package that can marshal itself to XML. If we want to be able to marshal to both formats, our problem is then reduced to converting the original Open-DIS PDU to a JAXB-generated PDU, and vice versa. Since all the fields and accessor methods have exactly the same names, it turned out to be easy to modify our code generation tool to write methods in the Open-DIS PDU that converts between JAXB-generated and Open-DIS objects. The process is shown graphically in figure 8.

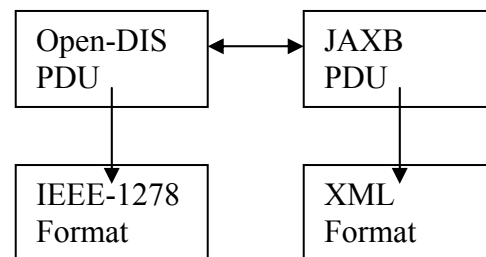


Figure 8: Marshalling to XML

An Open-DIS object can create a JAXB-DIS object with the same field values, and can convert a JAXB object to an Open-DIS object with the same field values. This allows us to marshal to either format.

This is, to be polite, a roundabout way of getting to the desired end state, but we were able to implement the code relatively quickly, and as a result are able to read and write DIS PDUs in an XML format.

2.5 Networking Support. The Java implementation includes example networking code to send and receive IEEE format DIS PDUs on a multicast socket in the `edu.nps.moves.examples` directory. The `EspduSender` and `EspduReceiver` classes illustrate sending and receiving entity state PDUs.

The C++ implementation relies on the HawkNL network library and also contains example code

for sending and receiving PDUs. Any networking package could be used in place of HawkNL, including WinSock or Berkeley sockets. The difficult part is getting the information into IEEE-1278.x format; once that is done the networking code is not extensive.

2.6 Community The DIS implementation is not complete, and it would be helpful to have the assistance of others in completing the library and adding new features. Doubtless bugs exist in the current Java and C++ implementations. Open source projects are more than a one-time code dump; they need a supporting infrastructure to maintain and improve the code, which is ultimately a social process.

To this end we have set up a site at sourceforge.net, the leading repository of open source projects. Sourceforge.net maintains a permanent site for CVS/Subversion, forums, mailing lists, code releases, mirrors, and the full spectrum of development tasks. The URL for the Open-DIS project is <https://sourceforge.net/projects/open-dis>.

3. Performance

Previous experience has shown that the performance of protocols that use TCP/IP UDP as transport is dominated by the number of packets per second that arrive on the network interface. Typically a TCP/IP stack generates an operating system interrupt for each UDP packet that arrives. The interrupt handling code has a higher priority than any user process, and the interrupt handling code is relatively CPU-intensive. All this can easily lead to a situation in which the network interface device driver, TCP/IP stack, and operating system are monopolizing the CPU to process packets rather than allowing the user-space process to handle parsing the protocol payload. Benchmarks that count the number of DIS UDP packets per second processed by a host are somewhat suspect; they often measure the tuning and efficiency of the operating system, TCP/IP stack, network socket options, and the characteristics of the sending process rather than any inherent characteristics of the DIS protocol implementation.

Nonetheless, we benchmarked a 3 Ghz Pentium 4 CPU running CentOS 5 Linux processing about 500 DIS UDP/multicast packets per second using the C++ implementation. Running

on a 2 Ghz, Intel Core Duo with Apple OS X v. 10.5, about 1,000 entity state PDUs per second were received and processed. This suggests users should be able to process perhaps a few hundred PDUs per second on a modern CPU, assuming that some of the CPU budget will be expended on other tasks such as dead reckoning, physics, graphics, etc.

4. Conclusions and Future Work

The Open-DIS library promises to simplify the implementation of new DIS-related software by removing licensing obstacles and barriers to entry. Use of the library should match well with the very long project lifecycles of DoD software, reduce duplicated effort, and focus programmers on adding new value rather than re-implementing old functionality yet again.

It is notable that we chose to use a self-defined XML dialect rather than XML schema. This choice was made in part because the XML schema could not easily represent the relationships between some DIS protocol fields. A system for annotating XML schemas with additional information might have been helpful in overcoming this problem.

The process of reading the DIS standard and creating the abstract description of the protocol is tedious and error-prone. It would be helpful for implementers if the protocol were defined in an unambiguous and computer-friendly format to begin with. This would allow computer language implementations to be created directly from the standard, rather than having a human read and interpret the standard to create the description. This would greatly lower the barriers to entry in creating implementations of protocol standards.

Further work can be done in completing the PDU implementations, and in creating supporting software such as data loggers, dead reckoning software, client-side finite state machines, enumerated types, and more. DIS enumerations are a particularly promising area for further work, since the DIS enumerations database is in the process of being converted to XML. This should facilitate the automated generation of enumeration data structures.

The Open-DIS code should provide a stable foundation for this work. An obvious extension would be to integrate Open-DIS with an HLA RTI to create an HLA-DIS gateway, perhaps

using the Portico open source RTI [xxx]. Open-DIS may facilitate integration with other networking standards, such as massively multiplayer online game engines.

1. IEEE Standard 1278.1-1995 (and revisions), Standards Committee on Interactive Simulation (SCIS) of the IEEE Computer Society, Approved September 21, 1995.
2. McGregor, Don, Brutzman, Don, Blais, Curtis, Arnold, Adrian, Falash, Mark, Pollak, Eytan: DIS-XML: Moving DIS to Open Data Exchange Standards, Spring Simulation Interoperability Workshop, 1997.
3. Sun Microsystems, <https://jaxb.dev.java.net>, downloaded February 1 2008.
4. Jeffrey Weekley, Don Brutzman, Anthony Healey, Duane Davis and Daryl Lee, "AUV Workbench: Integrated 3D for Interoperable Mission Rehearsal, Reality, and Replay" 2004 Mine Countermeasures & Demining Conference: Asia-Pacific Issues & Mine Countermeasures (MCM) in Wet Environments, Australian Defence Force Academy, Canberra Australia, 9-11 February 2004.
5. Web3D Consortium, <http://www.xj3d.org>, downloaded February 1, 2008.
6. Macedonia, Michael, Zyda, Michael, Pratt, David, Brutzman, Donald, Barham, Paul: *Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments*, IEEE Computer Graphics, September 1995, Vol. 15, No. 5, pp. 38-45.
7. Gamma, Eric, Helm, Richard, Johnson, Ralph, Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995 Addison-Wesley